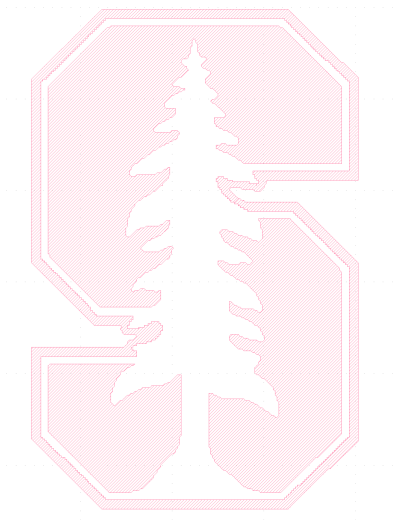
Creating a GDS mask file for metasurfaces

Department of Electrical Engineering, Stanford

October 12, 2020

Abstract

This package is intended for creating GDS mask files typically used to specify the geometry of a sample for a nanofabrication process. The focus is on meatasurfaces/metagratings, i.e. from a set of basic building blocks (unit cells, meta-atoms) the whole structure will be built by matching their individual phase response with a given in-plane phase distribution. The functions in the package provide a user interface in Matlab, along with some routines, while the actual GDS file generation and low-level operations are performed using Ulf Griesmann's (NIST) GDSII Toolbox.[1]



[1]The Toolbox can be downloaded from: https://sites.google.com/site/ulfgri/numerical/gdsii-toolbox

Contents

1 Introduction

There are two versions of the GDSII Toolbox[1], either of which can be used. However, the package may require two Matlab functions available only in v144. They can simply be copied to v99 with no issues. See Appendix A for their locations, as well as some notes on the potential bugs that might arise when using the Toolbox, though normally they shouldn't, and the ways to fix them. The use of v144 is recommended: for large files the write time is minutes versus hours for v99 and the end file size is also somewhat smaller. The Toolbox documentation explains how to get started and .gds file format specifics, and the main concepts are summarized here.

1.1 GDSII Toolbox Installation

To get started with using Ulf Griesmann's Toolbox, one needs to (a) add the folder with the corresponding version including all its subfolders to the Matlab's path (as always, same as with this package), and (b) to compile the .c files. For the latter, follow the following three simple steps from Matlab's command line:

1. run `mex -setup` to check the compiler used by Matlab,
2. navigate to the unzipped Toolbox folder using `cd` or Matlab's current folder panel,
3. run `makemex` to compile the files.

1.2 GDS Format Basics

To visualize the mask geometry, several free software options are available, the most popular of which is KLayout.[2] A typical file consists of *cells* that can exist either on its own or be a part of a hierarchy. The cell that's the highest in a hierarchy, called the *top* cell, is shown once the file is open and will be used for the manufacturing process. Cells are also known as *structures* and have a unique name assigned to each of them. The building blocks of a structure are called *elements*. They are nameless and contain, basically, either coordinates of a shape (or a set of thereof) or a reference to a specific structure. A reference carries a transformation (translation and rotation) to be applied to the referenced structure when it is being used to create a new structure. This allows us not to duplicate the full geometry information for identical structures. References can be either *structure* (`sref`) or *array* (`aref`) references, where the former produces a new structure with a single instance of an old one, while the latter gives their array on a rectangular grid. The functions in this package take advantage of `sref`'s only.

[2] klayout.de

2 Overview of the functions

This package consists of the following functions:

- `genGDS`
- `gen_gds_structure`
- `sref2boundary`
- `eldiff`
- `ellipse`
- `rectbox`
- `arc`
- `updatestruct`

Description of the functions is available through the `help` command. In this section we provide some details on each of them that will be relevant to the user.

2.1 genGDS

This is the actual function to by called by the user. It creates the GDS file of the structure and writes it on disk.

```
L = genGDS(phase_data, varargin)
```

Inputs. The first argument, `phase_data`, specifies the phase distribution map. Currently, only maps that are axially symmetric or invariant in one dimension (cylindrical lens) are supported. This means that we have to define the phase as a function of a single variable that we'll call `R`. `phase_data` can be defined in one of three formats:

1. two column matrix with `R` and the corresponding `phase` values in the first and second columns, respectively,
2. `struct` with two fields: `R` and `phase`,
3. `function_handle` that takes `R` (scalar or vector) as an argument and returns a `phase` value.

`phase_data` is an optional argument. If it is not provided or is empty, the function operates in the *test mode*. The test mode is useful to verify the appearance of the unit cells positioned side by side before generating the full file. Its operation depends on the *type* of the structure (see options below). In the case of a circularly symmetric structure, the test mode takes each meta-atom in the order they are provided and uses them to create rings starting from the center. For a cylindrically symmetric structure (invariant along one dimension), the meta-atoms are placed in line one by one, from left to right.[3]

[3]The size of the array can be set via the `Rmax` option, which will lead to the duplication of the unit cells next to each other $(u_1, u_1, u_1, ...u_n, u_n, u_n, ...)$ instead of cyclically $(u_1, u_2, u_3, u_1, u_2, ...)$, which in turn can be achieved by defining a special `phase_data` function. See Examples for an illustration of both cases.

The rest of the arguments to `genGDS` are Name/Value pairs of the following options, which can be provided in an arbitrary order. Some are optional, but duplicates will return an error. Non-empty default values are provided in [brackets], while the class of a variable is given in *italics*. Available options are:

- `type`, *char array*, [`'circ2cart'`]
  The type of the surface pattern. For axially symmetric patterns, the coordinate origin is in the center, while for cylindrical ones, it's in the bottom left corner of the structure. `type` can take one of the following values:
  - `'circ'`: circular metasurface on a polar grid, meaning that a rectangular unit cell will be bent into an arc. Uses the METAC algorithm[4] for GDS file structuring, which dramatically reduces its size, but increases the amount of memory required to render due to the deep internal referencing hierarchy. At present, there is no way of setting an upper bound on the number of referencing levels.
  - `'circ+'`: same as `'circ'`, but tries to adjust the positions of elements of a given unit cell structure to account for the distortions due to the curvature. Experimental. The optimal amount of the adjustment is potentially to explore. See Section 3.4 for a demonstration of the difference between the two options.
  - `'circ2cart'`: circular pattern on a Cartesian grid, i.e. boundaries of phase levels suffer from the staircase approximation.
  - `'cyl'`: cylindrical metalens. The direction of constant phase is chosen to be y. The algorithm is to (i) predefine 'stripes', which are arrays of `sref`'s to a given unit cell, of width equal to period in the x dimension and height equal to the metalens height (`Rmax(2)`); (ii) moving one unit cell width at a time in x direction create an `sref` to a stripe that has the closest `phi0` value to the one of `phase_data(x)`. Referencing one stripe at a time, instead of, say, doubling their number at each step, allows to have different periods in both x and y, and would be non-trivial to implement due to variable widths of the Fresnel zones. For a deflector, one could define the whole zone that covers $[0, 2\pi]$ phase and refence it.
  - `'cyl2cart'`: cylindrical metasurface/metalens, where each pixel is a `sref` to the corresponding meta-atom. This will give a larger file size, compared to `'cyl'`. Morover, `'cyl'` should not suffer as much from the memory usage increase thanks to shallow referencing depth, as compared to `'circ'`, where it can approach 20 for large-scale metasurfaces. This said, generally, `'cyl'` is preferred.

[4]She, A., Zhang, S., Shian, S., Clarke, D. R., & Capasso, F. (2018). Large area metalenses: design, characterization, and mass manufacturing. *Optics Express*, 26(2), 1573–1585. DOI:10.1364/OE.26.001573

- `shading`, `'holes'` or `'posts'`, [`'posts'`]
  specifies whether the shaded region of the GDS file should correspond to the holes, or posts (areas occupied by the material) in the design.
- `fname`, *char array*, optional
  name of the output file. Adding the format ('.gds') is optional. Prepend the name with '!' to overwrite a potentially existing file of the same name. If none is given, no file is written on disk and the user has to make sure to fetch the output, which later can be saved by calling `write_gds_library(L, fname)`.
- `geom`, *cell array*, required
  a cell array (or numerical matrix/struct for a single unit cell), where each cell defines a unique meta-atom geometry.[5] A unit cell can consist of one or multiple sub-elements, which can be defined as one of (i) numerical or logical array representing a binary image, (ii) two-column matrix of x and y coordinates tracing a polygon, (iii) `struct` with the following fields:
  - `fun`, name of the function to use to get the boundary, either `'ellipse'` or `'rectbox'` to use the corresponding functions, or a custom function handle or name. See after this list for more details;
  - `r`, two element vector with semi-axes radii if `fun` = `'ellipse'` and width and height pair if `'rectbox'`. Can be a scalar for a circle or square, respectively;
  - `c`, coordinates of the element's center within the unit cell;
  - `angle`, angle of rotation (in degrees) around `c`.

  Instead of using `'ellipse'` or `'rectbox'`, one has an option of providing a custom function as a `function_handle` (`@myfun`) or a name (`'myfun'`). In this case, field `r` should keep a cell array of the first $N_{arg}-1$ arguments and `c` should still be a pair of coordinates of the center and the final argument of the function. Naturally, `angle` can still be defined as any of the arguments, except the last one. The reason to require `c` as a separate field if because it's coded to be used with `'circ'` and `'circ+'` layouts.

  Different cells of `geom` (meta-atoms) can have be defined differently, but subelements of a given cell have to be of the same type. For numerical inputs, they should be gathered in a cell array, for a struct – a simple array.[6]

[5] Doesn't have to be unique for the code to run, but from a physical view point, identical geometries give identical phase differences, and if a duplicate `geom`/`phi0` pair is provided, the code will use the first instance only.

[6] Note that for custom geometry functions, when one defines `struct('c', {[0 0], [1 1]})` in Matlab, instead of a potentially expected outcome of a struct with a cell array in the `c` field, one gets a *struct array*, each with its pair of coordinates, which is not what's desired here. It is laborious to correct it automatically in the general case where there are multiple such elements per unit cell.

- `period`, *numerical array*, depends
  generally, a two column vector of periods in x and y directions, with rows corresponding to respective elements in `geom`, and required, but for image inputs can be omitted – their sizes will be used instead, i.e. one pixel is one `uunit` large. Can be a single row if all basic unit cells have the same periods, and single column for a square unit cell.

- `phi0`, *numerical array*, depends
  numerical array of characteristic phase responses in radians of unit cells in `geom`. Optional only in *test mode*.

- `uniform_boundary`, *logical*, [`false`]
  defines the `equal_step` parameter of `ellipse` function for the case when `geom` is defined as a struct (and `geom.fun` is `'ellipse'`). It uses the `interparc` function[7] in order to select points on a curved surface with equal separation along the trajectory. This option can produce smoother-looking shapes, but also can slow down `genGDS` a little if called lots of times.[8]

- `A`, *cell array*, optional
  cell array of the same size as `geom` (or just a matrix for a single meta-atom) of either full or sparse square (usually logical) matrices that represent the parent-child dependencies between object boundaries and hole boundaries. For a unit cell defined as either a cell array of paths, or an array of `struct`, `A` gives an adjacency matrix between the boundaries. For unit cells that are defined through an image, `A` is calculated automatically with `bwboundaries`. Therefore, its definition follows the one in the output of `bwboundaries`. If the matrix has a non-zero element (`true`) at the position (i, j), it means that i^{th} boundary (child) will be subtracted from j^{th} (parent). If there is no parent/child elements for a given unit cell, its `A` can be either empty or all zeros.

- `n`, *numerical/cell array*, optional
  array (size equal to `geom`'s) of integer numbers of objects in the unit cells. This is `bwboundaries`'s definition that does not make much sense here. What to do with it, depends on the way `geom` is defined:

  1. `geom{i}` is a binary image: `n` is not required, as it's calculated automatically using `bwboundaries`, along with `A`;
  2. It's a `struct` and `A` is constructed manually: in this case, `n` would be equal to the number of columns (or rows, since `A` is a square matrix) in the corresponding `A`, as the user specifies all the subtractions needed. In this case, the user does not have to explicitly provide `n`.

[7] Available at: https://www.mathworks.com/matlabcentral/fileexchange/34874-interparc
[8] But were to hurry?

3. It's a cell array of boundaries: if `A` was defined manually, `n` similarly can be omitted, if, however, it was obtained using `bwboundaries`, n has to be fetched from the eponymous output variable, since, in general, `size(A,2)` would be larger than `n`, as `bwboundaries` first puts objects (parents), then holes (children), and we only need to loop over the objects.

- `uunit`, *numerical scalar*, $[10^{-6}]$
  user units as a fraction of a meter. Default – microns.
- `dbunit`, *numerical scalar*, $[10^{-11}]$
  database units as a fraction of a meter, used for storing the data. In other words, it determines the resolution. Keep in mind, that GDS stores the positions as a 4-byte *integer*, so there's a trade-off between the resolution and the maximum size (largest coordinate value) of the structure. The number of significant digits after the decimal marker in a length specified in user units is given by `log10(uunit/dbunit)`.
- `Rmin`, *numerical scalar*, [0]
  inner radius of the metasurface. For a circular metasurface it means that we have a void in the center of radius `Rmin`, which is automatically shaded if `shading` parameter is `'holes'`. This might be useful to avoid regions of extreme curvature. For a cylindrical metasurface, it shows a shift of coordinates in x-direction, which is not useful and should be left 0.
- `Rmax`, *numerical scalar or two element vector*, depends
  outer radius of the metasurface. For a cylindrical (rectangular) metasurface, it is a two-element vector of the upper-right corner of the device. Optional only in test mode.
- `ispost`, *logical*, [`true`]
  specifies whether the provided boundary in `geom` encircles areas of the material, or, if an image is given, whether non-zero pixels mark the (higher-index) material.
- `pts`, *numerical scalar*, [64]
  number of points per single boundary generated by the `ellipse` function (if it's used at all), i.e. its `t` argument.
- `bgd_tiling`, *numerical array*, [1]
  size of grid to break the 'background' of a unit cell into. Can be a scalar for a square grid. If we need to produce a "full unit cell", [9] `gen_gds_structure` takes the 'background', which corresponds to the metasurface unit cell,

[9] "Full unit cell" refers here to the case when unit cell is shaded with its borders and thus depends on the curvature. This is the case when either `ispost` is `true` and `shading` is set to `'holes'`, or `ispost` is `false` and `shading` is `'posts'`. See examples for some illustrations of these options.

which is either a rectangle or an arc, with filling fraction equal to one, and subtracts from it all subelements whose boundaries are provided.

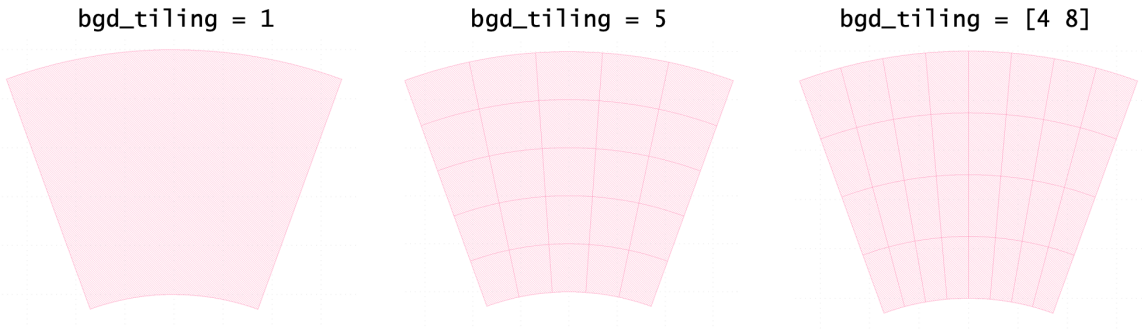


Figure 1: Demonstration the `bgd_tiling` option impact on an arc section.

- `layer`, *numerical scalar*, [1]
  layer number in the GDS file (all structures are on the same one, though they don't have to be). Not really important.
- `label`, *struct*, optional
  parameters of the label to be created. A struct with the following fields:
  - `string`, text to be written;
  - `height`, maximum letter height in user units (`uunit`);
  - `max_feature`, largest feature i.e. the square diameter for the checkerboard pattern;
  - `location`, position on the label, either a two-coordinate vector of the lower left corner, or one of the following options:
    - `'ne'`: north-east, upper right corner,
    - `'nw'`: north-west, upper left corner,
    - `'out'`: the text ends in the upper right corner right outside the metasurface region for cylindrical metasurfaces (rectangular device) and inside for circular (corners are empty),
    - `'center'`: text hovers above the middle point of the device.

    Optional, `'ne'` by default;
  - `mirror`, boolean, whether to position a copy the text upside down on the opposite side of the device. Optional, `true` by default;
  - `angle`, angle to rotate the text in degrees. Optional, 0 by default.

Output. `genGDS` has a single output – a variable of class `gds_library` of the GDSII Toolbox. Normally, it is not used directly, since if the `fname` option

is provided, the data is written into a .gds file of that name. However, in some cases, one might still be interested in keeping the `gds_library` object for some further manipulations, and write it later.

2.2 gen_gds_structure

This function is called by `genGDS` to transform the unit cells provided as a `geom` variable into the instances of the `gds_structure` class, which will be further used as building blocks by `genGDS` to generate the structure according to the specified patterning option `type`. See `help gen_gds_structure` and comments inside for more details on its operation, though understanding how it works is not necessary to use `genGDS`. One important fact is that `gen_gds_structure` can be used in one of two ways. For circular patterns, namely, `type` option being `'circ'` with full unit cell shading[10] or `'circ+'` with either shading, it is called for each ring to recreate the appropriate unit cell (according to the phase function) with the corresponding curvature. The number of basic cells is thus equal to the number of rings. In all other cases, `gen_gds_structure` creates the basic unit cells from `geom` as `gds_structure` objects in the beginning of the `genGDS` call to be later referenced with an `sref` object. The number of basic cells is thus equal to the length of `geom`.

2.3 sref2boundary

An `sref` object keeps the name of a `gds_structure`, as well as a transformation (translation and/or rotation). This function takes an `sref` along with the referenced object and returns a new one, with the transformation performed.

2.4 eldiff

This function is a simple implementation of taking a difference between two elements, or paths, to substitute GDSII Toolbox's `poly_bool` which can thow errors due to structures with a hole being created. `poly_bool` uses the Clipper C++ library and works best with cases in Fig.2(a), while fails when one object is completely encircled by another [Fig.2(b)]. This can be remedied by increasing the `bgd_tiling` parameter, which breaks the unit cell background (encircling object) into a grid of smaller adjacent squares (cylindrical lens) or arcs (circular) [Fig.2(c)].

`genGDS` will first try to use `eldiff`, because fully overlapping regions is a more frequent occurrence, particularly with `bgd_tiling` being equal to one. If it fails, due to the background being more than one path (`bgd_tiling` > 1) or due to some input it can't handle, `poly_bool` will come into play. For `poly_bool` the

[10] See footnote 9 on "full unit cell".

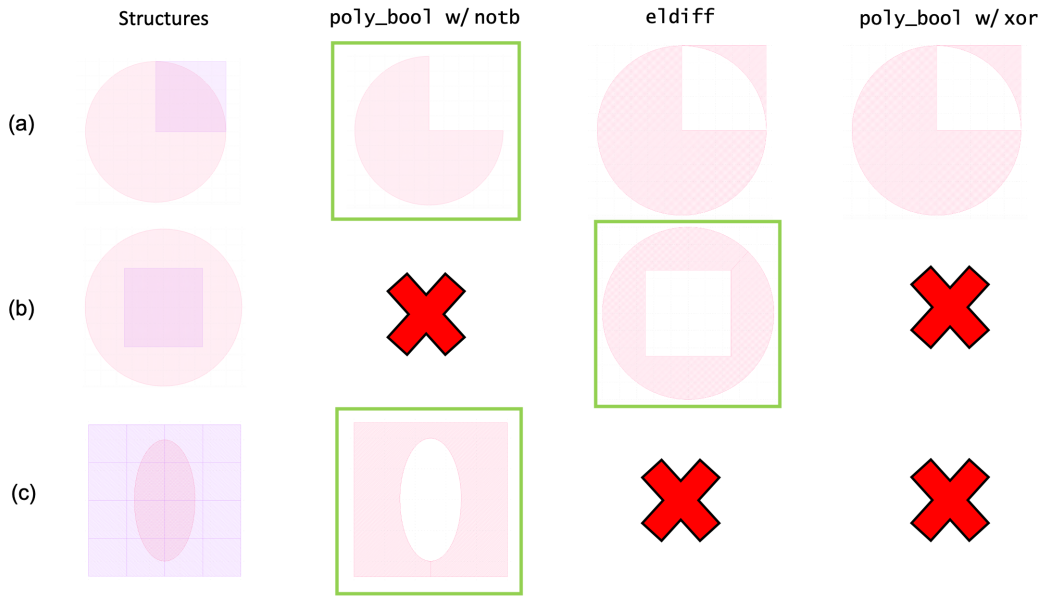


Figure 2: (a) Intersecting structures. Expected result is given by `poly_bool` with the option `notb`. Simplistic algorithm of `eldiff` fails, producing a structure equivalent to the `xor` operation. (b) Fully overlapping structures. `poly_bool` fails due to a hole being created, while `eldiff` gives the expected result (notice the 'seam' in the upper-right corner. (c) Fully overlapping structures with 'tiled' background. `poly_bool` produces the correct result, when the size of tiles is small enough, but not smaller to be completely covered by another object. `eldiff` is not suitable for this case, and `gen_gds_structure` selects `poly_bool` automatically.

size of a 'tile' has be small enough, not to create a hole, but not too small, not to lead to one being removed completely. This is a bit of an artificial limitation that could potentially be an issue. A better tiling scheme could help, or some better understanding of the Clipper library and modifications to `poly_bool`.

2.5 ellipse

Generates a two-column matrix `pt` with x and y coordinates of a contour of an ellipse. It has an optional parameter, `equal_step`, which corresponds to the `genGDS`'s `uniform_boundary` option. This allows for the vertices of the polygon that describe an ellipse to be positioned with equal intervals between each other along the curved arc trajectory, while the default behavior (`uniform_boundary` = `false`) uses equal angles of the arc sections. For this

functionality, `ellipse` takes advantage of the `interparc` function that can be obtained from https://www.mathworks.com/matlabcentral/fileexchange/34874-interparc.

2.6 rectbox

`rectbox`, like `ellipse` returns coordinates, but of a rectangular box. Size of the output matrix is 5×2, where the first four rows are the coordinates of the four vertices, and the fifth is identical to the first to form a closed path.

2.7 arc

`arc` returns a boundary of an arc element (see Fig.2.1) by calling `ellipse` twice.

2.8 updatestruct

A general-purpose function that takes two Matlab's `struct` objects and updates the fields of one with those of the other, with a few options how to deal with the fields not present in the `struct` being updated.

2.9 gds_examples

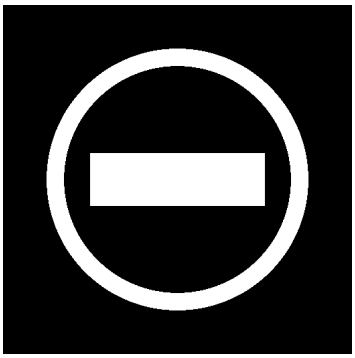
Examples from Section 3 of this manual as a Matlab Live Script.

3 Examples

3.1 Defining geometry as an image

There are three ways to specify the unit cells geometry with `genGDS`. Let us start with the most general one: a binary image. A binary image is an image of ones and zeros, or, more generally, zero and non-zero pixels. In the metasurface context, we are interested in ones being, usually, the higher-index material, and zeros – the lower-index one. The input to `genGDS` will be passed through the `round` function to treat any grayscale pixels that might be left from topology optimization. Then, the `bwboundaries` function (Image Processing Toolbox) will be used to find all boundary regions in the image. It supports both numerical and logical matrices and distinguishes only between zero and non-zero pixels. For example, `imread` loads this PNG file as a `unit8` matrix of 0 (black) and 255 (white) values.

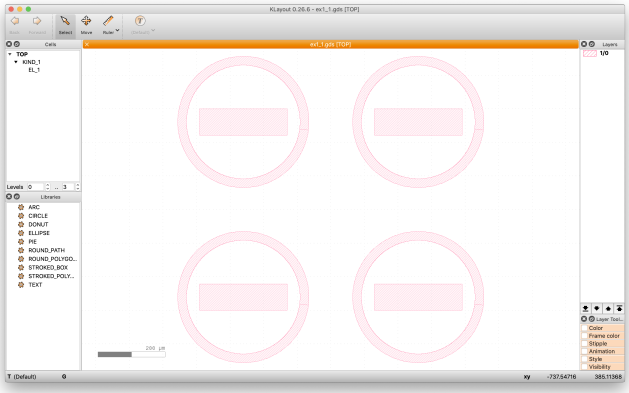
```
BW = imread('example.png');
imshow(BW.')
```



Now, we can write a basic .gds file (exclamation point in front of the file name is a signal to the GDSII Toolbox to overwrite the file if it exists):

```
genGDS('fname','!ex1_1','geom',BW);
```

Inside `ex1_1.gds`, we will see (screenshot of the KLayout viewer):



We have four of them, because the default type option is circ2cart, so they form the smallest circle approximated with rectangular pixels. Notice the seam on the right. It is there because the gds file format requires a closed contour with no holes, so the inner and the outer boundary have to be connected. The function takes care of the relationship between the boundaries automatically.

The default value for the `shading` parameter is `'posts'`, i.e. regions with non-zero elements are colored. Let's change the shading to the opposite one – `'holes'`:

```
genGDS('fname','!ex1_2','geom',BW,'shading','holes');
```

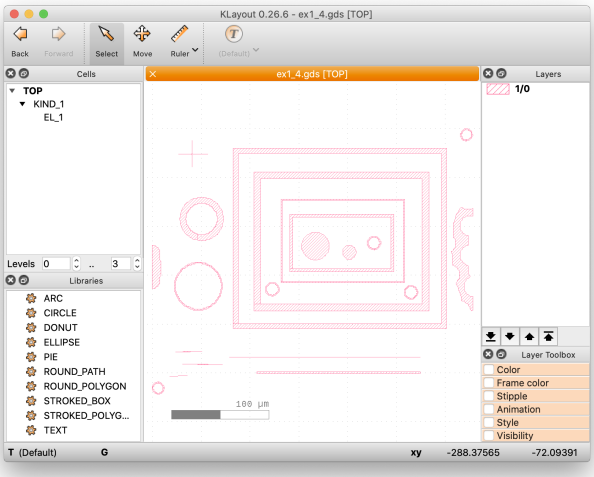
Notice again the line connecting the inner and outer boundaries.

There is an option called `bgd_tiling` that divides the background into smaller adjacent regions. This is also a way to force `genGDS` (actually, `gen_gds_structure`) to use GDSII Toolbox's `poly_bool` over `eldiff` (see the previous Section for more details). However, for many cases, including this one, it is not necessary, but for the sake of demonstration, we include it here. This example divides the unit cell into a 4×4 grid. This option requires the 'tiles' to be no smaller than the smallest element. Fox example, it will fail with `bgd_tiling` equal to 5, since in this case, the rectangle in the middle removes a whole tile (thus creating a 'hole' of more than it's size).

```
genGDS('fname', '!ex1_3', 'geom', BW, 'shading', 'holes', '
    bgd_tiling', 4);
```

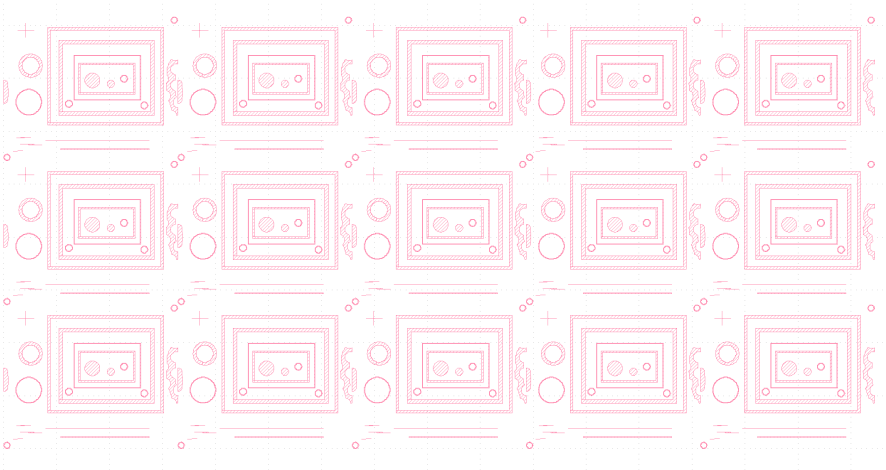

As a more challenging and interesting example, let's try one from Matlab's test images:

```
BW = imread('blobs.png');
genGDS('fname', '!ex1_4', 'geom', BW.','type','cyl');
```



We can easily show a grid of images.

```
P = size(BW.'); % period, same value as the default one, need for
    the dimensions
Nx = 5; % number of repetitions in the x dimension
Ny = 3; % in y
genGDS('fname', '!ex1_5', 'geom', BW.', 'period', P, 'type', '
    cyl2cart', 'Rmax',[Nx Ny].*P);
```

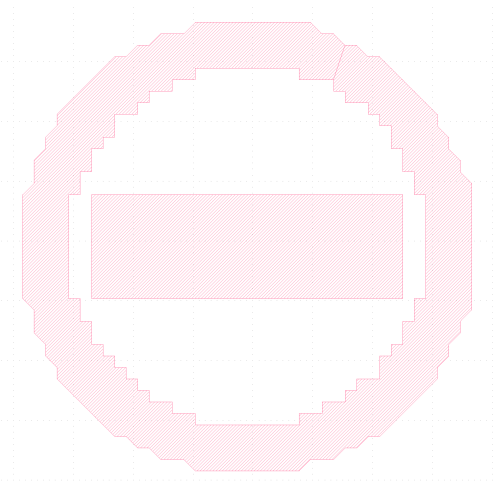


Note that for a single element before we've left the default `type` option `circ2cart`, which means the circular arrangement of the elements, and our was in the center. While for this example, we want to have a 2D array, therefore, we change it to `cyl2cart` (`cyl` would work exactly the same).

On a final note, keep in mind that the image resolution will play a role in the smoothness of the mask. `bwboundaries` returns the coordinates of the border non-zero pixels, which are rectangles, but GDS software just connect them with a line and shades one side, which might cause ragged shapes.

```
BW = imread('example.png');
% reduce size to the 10% of the original image
BW = imresize(BW, .1);
genGDS('fname', '!ex1_6', 'geom', BW);
```

What we get in the mask file is:



3.2 Defining geometry as a boundary

A boundary is usually fetched from an image, or generated by some custom function. For instance, to get the same result as `ex1_1`, we could do

```
BW = imread('example.png');
[b,~,n,A] = bwboundaries(BW);
genGDS('fname', '!ex2_1', 'geom', b{1}, 'period', size(BW));
```

Here, we trace a bar from `example.png` provided as an Nx2 matrix. The period is now a required parameter. Multiple boundaries have to packed in a cell array, which in turn has to placed in a cell of another, to mark a metasurface unit cell consisting of multiple boundaries. For example, to transfer the full image to the mask, the command would be

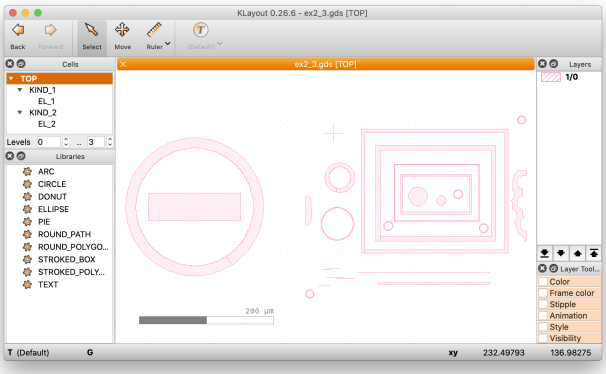
```
genGDS('fname', '!ex2_2', 'geom', {b}, 'n', n, 'A', A, 'period',
    size(BW));
```

Note that an adjacency matrix is needed to re-create the image, since boundary #3 (inner circle) has to be subtracted from #1 (outer circle).

Let's try putting the second image side-by-side.

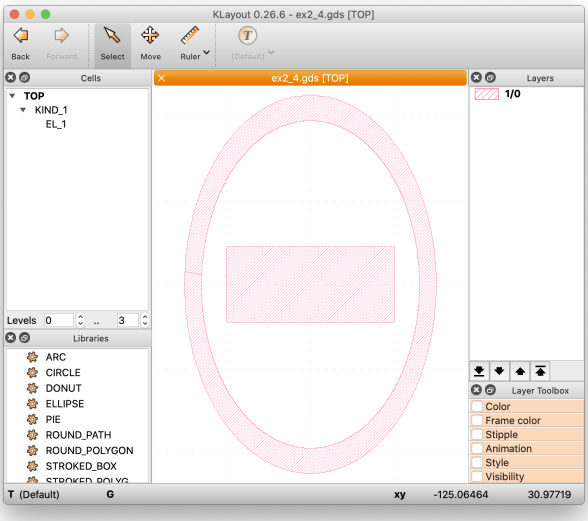
```
BW2 = imread('blobs.png');
BW2 = BW2.'; % first dimension x, second y, different from imshow
% make the images have same height
BW = imresize(BW, size(BW2,2)/size(BW,2));
[b,~,n,A] = bwboundaries(BW); % recalculate for the new size
[b2,~,n2,A2] = bwboundaries(BW2);
genGDS('fname', '!ex2_3', 'geom', {b,b2}, 'n', [n n2], 'A', {A,
    A2}, 'period',[size(BW);size(BW2)], 'type', 'cyl2cart');
```

The output is now:



That this is not enough for real cases, since we're not specifying a real period here. For example, let's consider a rectangular period that doesn't have the side ratio of either image:

```
BW = imread('example.png');
[b,~,n,A] = bwboundaries(BW);
P = [200 300]; % e.g. microns
% transform pixel number to coordinates
xGrid = linspace(0, P(1), size(BW, 1)+1);
% offset to the center of a pixel
xGrid = xGrid(2:end).' - xGrid(2)/2;
yGrid = linspace(0, P(2), size(BW, 2)+1);
yGrid = yGrid(2:end).' - yGrid(2)/2;
for i = 1:length(b)
    b{i} = [xGrid(b{i}(:,1)), yGrid(b{i}(:,2))];
end
genGDS('fname', '!ex2_4', 'geom', {b}, 'n', n, 'A', A,'period',P,
    'type','cyl');
```

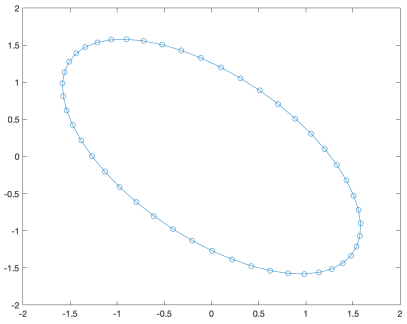


The geometry was scaled proportionally to match P. If we didn't put `b` inside the cell of it's own, the three components would be considered as different meta-atoms.

3.3 Defining geometry as a struct

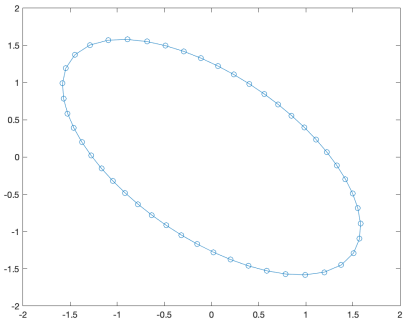
Whereas before the focus was on freeform shapes, this section will deal with the cases where a boundary can be approximated with a simple (or not so simple) shape by defining geometric parameters as arguments to a function that produces a boundary. There are two functions that are included in the package: `ellipse`, for elliptical shapes, and `rectbox` for rectangles.

```
% args: semi axis x, y, center, number of points, angle of
    rotation
pt = ellipse(1,2,[0 0],48,45);
plot(pt(:,1),pt(:,2),'-o')
```



There's an additional optional last argument called `equal_step`, which, if set to true, uses the `interparc` third-party function (available from here) to distribute the points unifromly along the boundary.

```
% args: semi axis x, y, center, number of points, angle of
    rotation, equal_step
pt = ellipse(1,2,[0 0],48,45,true);
plot(pt(:,1),pt(:,2),'-o')
```



`rectbox` works in much of the same way. Let's define a structure similar to the one of `example.png`, using these two functions. It consists of two circles and one rectangle. Note that number of points and `equal_step` are to be defined not in individual `struct`'s, but as `genGDS` arguments and will be applied to all shapes generated with `ellipse`. The definition of an adjacency matrix is critical here, as the inner circle has to be subtracted from the outer. The center coordinates are defined with respect to the middle point of the unit cell.

```
% outer circle, 'r' can be a scalar here
circ1 = struct('r',[1 1],'c',[0 0],'fun','ellipse','angle',0);
% inner circle
circ2 = struct('r',.8,'c',[0 0],'fun','ellipse','angle',0);
box1 = struct('r',[1.2 .3],'c',[0 0],'fun','rectbox','angle',0);
% doesn't have to sparse, but it's easier to view
A = sparse(false(3));
% inner circle is in position 2 of the array below, outer - 1
A(2,1) = true;
genGDS('fname', '!ex3_1', 'geom', [circ1,circ2,box1], 'period',
    2.5, 'pts', 64, 'uniform_boundary', 1, 'A', A);
```

Our shapes are assigned to the same structure, as they are not siloed into different (Matlab's) cells. This has default values for `ispost`, which is `true`, meaning that the encircled regions correspond to material (higher index one at least), and `shading`, which is `posts`, i.e. material as well. The opposite mask can be obtained by flipping one of these parameters' values.

An important thing this enables us to do is to define custom functions (see manual for details). Here's an example:

```
N = 100; % number of points
theta = linspace(0, 2*pi, N).';
myfun = @(r,x,c) [(r(1)+x*sin(10*theta)).*cos(theta) + c(1), (r(end
    )+x*sin(10*theta)).*sin(theta) + c(2)];
s = struct('r', {{[10 5], 1}}, 'fun', myfun);
genGDS('fname', '!ex3_2','geom',s,'period',12,'type','cyl','Rmax'
    ,12);
```


The ‘`r`’ field has a special meaning now: it is a cell array of all arguments, whatever they may be, in the order they appear in the function definition, except for the last one, which is the coordinates of the center, `c`. `gen_gds_structure` will run something like `myfun(s.r{:}, s.c)`. Notice that we can omit fields `angle` and `c` to have zeros by default. However, be careful, Matlab will fail to concatenate `struct`'s with unequal set of fields. Notice also the important set of extra squiggly brackets to prevent Matlab from unpacking the cell array. This can be also achieved with dot assignment, e.g. `s.r = {1,2,3}`.

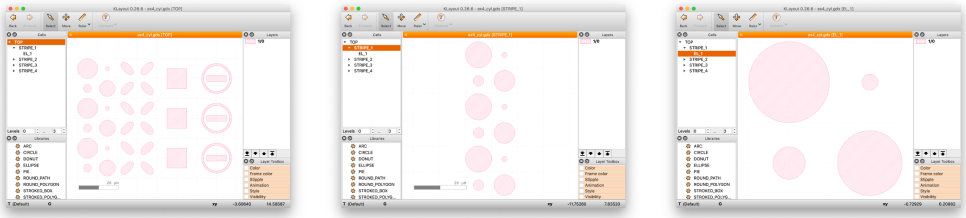
```
% array of two structs, not what we want
s = struct('r', {[10 5], 1}, 'fun', myfun);
```

3.4 Metasurface layouts

This example demonstrates the role of the type parameter. To start off, let's define a few different unit cells (four, actually):

```
ucells = cell(1,4);
% Unit cell #1: four circles of different size
ucells{1} = struct('r',{1,5,2,4},'c',{[5 5],[-5 5],[-5 -5],[5
    -5]},'angle',0,'fun','ellipse'); % 1x4 struct array
% Unit cell #2: four ellipses with an angle
ucells{2} = struct('r',[2,4],'c',{[5 5],[-5 5],[-5 -5],[5 -5]},'
    angle',{-45,45,135,-135},'fun','ellipse'); % 1x4 struct array
% Unit cell #3: simple square
ucells{3} = rectbox(10);
% Unit cell #4: image
ucells{4} = imread('example.png');
P = 20; % period (square, for simplicity)
```

```
genGDS('fname', '!ex4_1','geom',ucells,'period',P,'type','cyl','
    Rmax',P.*[4 3]);
```



Visually, both `cyl` andc `cyl2cart` are identical. The difference is only in internal structure: `cyl2cart` defines pixels as individual references, `cyl` defines 'stripes' (middle image) as a vector of references to a basic cell (right image), then references these stripes to build the mask (left image).

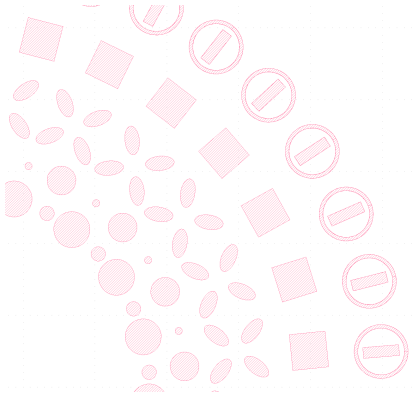
We can define a deflector in a number of ways by specifying the phase gradient function. One would be:

```
ncells = 4; % number of different unit cells per zone
phi0 = linspace(0,2*pi,ncells+1);
phi0 = phi0(1:ncells);
P = 20; % unit cell period
rep = 3; % number of repetitions of each unit cell
mult = 11; % number of zones to write'
mult_y = 4; % number of cells in y (invariant) dimension
Rmax = [rep * P * numcells * mult, P * mult_y];
phase_data = @(R)(mod(floor(R/P/rep),ncells)+1/2)/ncells*2*pi;
genGDS(phase_data,'fname','!ex4_2','type','cyl','Rmin',0,'Rmax',
    Rmax,'geom', ucells, 'phi0', phi0,'period', P);
```

As for the circular metalens, the options are (see Section 2 for more details):

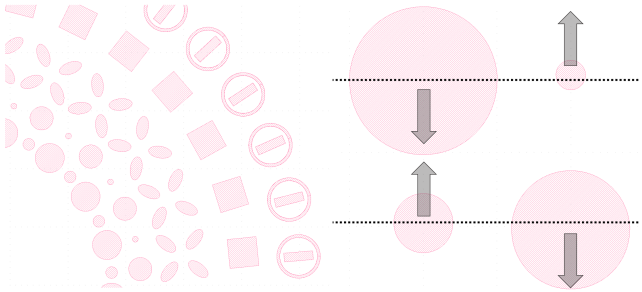
circ

```
genGDS('fname','!ex4_3','type','circ','Rmin',50,'geom',ucells,'
    period',P);
```



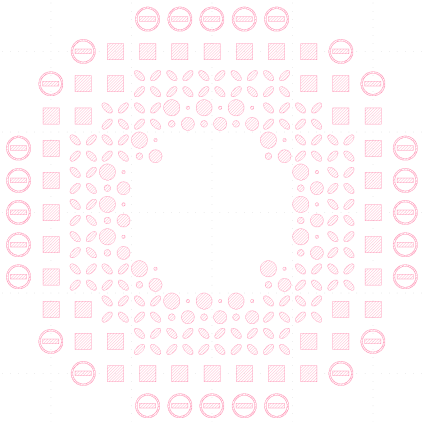
circ+

```
genGDS('fname','!ex4_4','type','circ+','Rmin',50,'geom',ucells,'
    period',P);
```



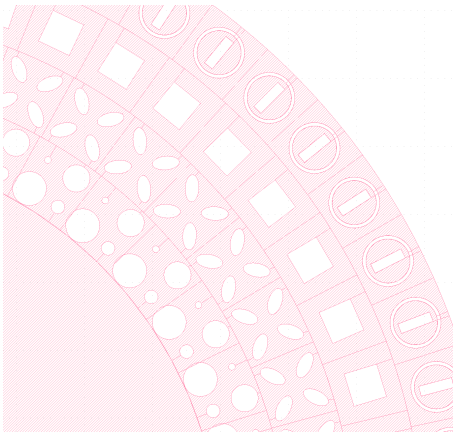
circ2cart

```
genGDS('fname','!ex4_5','type','circ2cart','Rmin',50,'geom',
    ucells,'period',P);
```



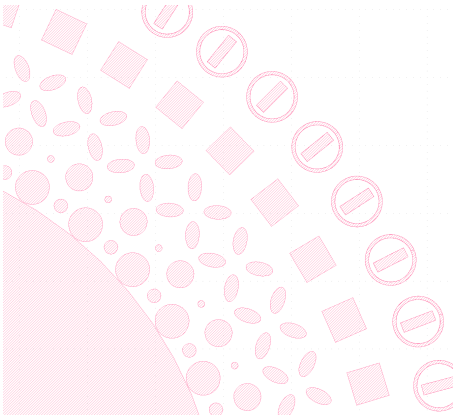
Inverse shading adds a hole in the middle for `circ` and `circ+`, while for `circ2cart` it doesn't, the idea being that the curvature is not an issue and we can keep `Rmin` zero:

```
genGDS('fname','!ex4_6','type','circ','Rmin',120,'geom',ucells,'
    period',P,'shading','holes');
```



We can flip the unit cell polarity, i.e. in the case of the image, 1's mark air holes with `ispost` set to `false`:

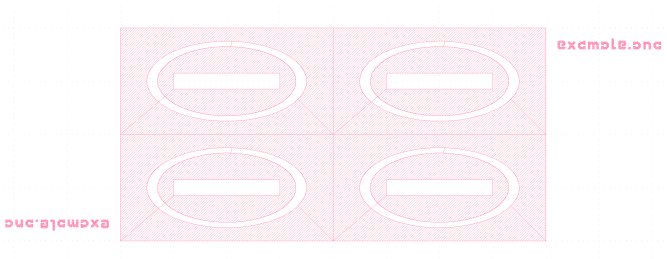
```
genGDS('fname','!ex4_7','type','circ','Rmin',120,'geom',ucells,'
    period',P,'shading','holes','ispost',false);
```



3.5 Creating a label

A label with the name of the structure positioned nearby is a useful reference during an experiment. For it to be easily resolvable, but also manufacturable, it is automatically superimposed with a checkerboard pattern. Let's see how it works and explore a few options.

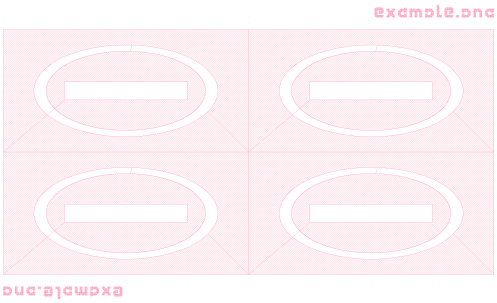
```
BW = imread('example.png');
clear opts
opts.fname = '!ex5_1'; % filename
opts.geom = BW; % geometries
opts.period = [20 10]; % size of the unit cell
opts.type = 'cyl'; % same result as with cyl2cart
Nx = 2; % number of prepetitions in x
Ny = 2; % number of prepetitions in y
opts.Rmax = opts.period.*[Nx Ny];
% change polarity to see the unit cell outline
opts.ispost = false;
opts.label.string = 'example.png'; % what to write
% height of letters in the user units (uunit)
opts.label.height = 1;
% largest feature (square diameter) of the checkerboard
opts.label.max_feature = 0.08;
opts.label.location = 'ne'; % top right corner, north-east
% duplicate the label on the opposite side upside down
opts.label.mirror = true;
opts.label.angle = 0; % text rotation
genGDS(opts);
```



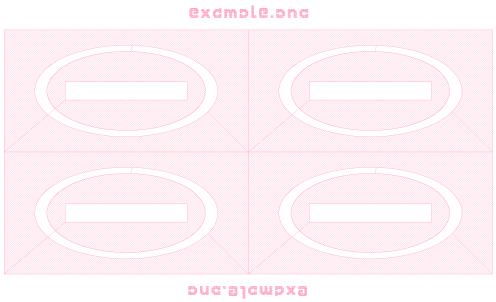
If we zoom into the label, we will see the pattern:

Other location options are available, such as, `nw` (north-west) for upper-left corner

or `out` to put it right outside the box in the corner (inside for circular structures, as corners are empty)



or `center` to position the text right above (below for mirrored) the structure:

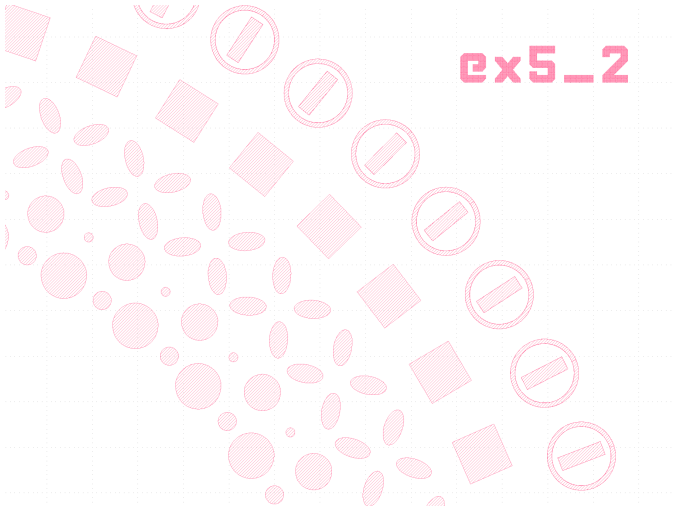


In addition, position can be a two element vector with x and y coordinates of the text bottom left corner.

Finally, one example with a circular metasurface. Using the same unit cells as before

```
ucells = cell(1,4);
% Unit cell #1: four circles of different size
ucells{1} = struct('r',{1,5,2,4},'c',{[5 5],[-5 5],[-5 -5],[5
    -5]},'angle',0,'fun','ellipse'); % 1x4 struct array
% Unit cell #2: four ellipses with an angle
ucells{2} = struct('r',[2,4],'c',{[5 5],[-5 5],[-5 -5],[5 -5]},
    'angle',{-45,45,135,-135},'fun','ellipse'); % 1x4 struct
    array
```

```
  % Unit cell #3: simple square
  ucells{3} = rectbox(10);
  % Unit cell #4: image
  ucells{4} = imread('example.png');
  label = struct('string','ex5_2','height',8,'location',[150 150],'
      max_feature',0.08);
genGDS('fname','!ex5_2','type','circ','Rmin',120,'geom',ucells,'
    period',20,'label',label);
```



A Notes on the GDSII Toolbox

There are two required functions that are available only in the GDSII Toolbox v144:

1. `Elements/gdsii_boundarytext.m`
2. `Basic/funcs/poly_rotz.m`

The first one is required to create a label – it automatically translates alphanumerical characters into GDS structures by tracing their outline. Inside, it calls `poly_rotz.m`, which is a simple matrix rotation operation:

$$\begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \tag{1}$$

Bugs that came up during development, but not testing for this manual:

- `Basic/@gds_element/poly_bool.m`
  `poly_bool` is called by `gen_gds_structure` if `eldiff` has failed to subtract one path from another and during the label creation taking the intersection of the text the checkerboard pattern. When taking the difference, an error can be created when a path with a hole is created. Sometimes (not too often though), the output geometry looks correct. To check it, change `error` to `warning` at Line 82 of the file and inspect the results using a layout viewer.
- (v99 only) `Basic/gdsio/gdsii_boundary.m`
  Line 70: substitute `length(xy)` with `size(xy,1)`
  Arises if you attempt to have a path that consists of a single point.
- (v99 only) `Basic/@gds_element/set.m`
  Line 7: change ! = to ∼=